Laboratory 2: Planning and Navigation

Drini Kerciku — 1004750780
Tian Yu — 1004750780
Ben Natra — 1005117366
Smit Patel — 1004734497

March 06, 2023

1 Introduction

An RRT and RRT* subroutines are implemented in python to find optimal paths through the Willow garage maze and a custom map set up in out laboratory in Myhal. In the following sections, we briefly describe the modules used for planning in terms of collision detection, trajectory rollout and rewiring of nodes.

2 Report

2.1 Collision Detection

The **checkCollision(...)** function takes a series of way-points that are part of a trajectory to be tested for collision. Every location is mapped to the global reference frame based on the '.yaml' configuration file for the map and discretized through **point_to_cell(...)** called inside **points_to_robot_circle(...)** function - a range of coordinates is generated to create an occupancy grid at the robots location through the **disk(...)** method per way-point. Finally, each point in the occupancy grid of the robot at each location is used to check whether it is occupied by an obstacle or not, and the result is returned accordingly.

2.2 Simulate Trajectory

simulate_trajectory(...) finds the trajectory that will nominally drive the robot from the start location to the end location nominally through the use of **robot_controller(...)** and **trajectory_rollout(...)** functions. Given the open ended problem for selecting the control inputs for the task, **robot_controller(...)** searches for a feasible control pair within the defined time horizon through a grid of possible input combinations of linear and angular velocities - a pair that drives the agent the closest to the endpoint is selected with the aid of **trajectory_rollout(...)** function calls. The trajectory of the returned inputs is simulated again and it is returned up to the point that is closest to the target based on our debug from run to run during development.

2.3 RRT Planning

The algorithm presented in class for RRT was closely followed in our implementation along with the structure provided in the skeleton code. A random point is sampled every iteration such that for the Willow garage map it is sampled within hardcoded bounds of the garage map to avoid generating paths in through the open space and around its exterior. Using the **cKDTree** object and its methods imported from the **scipy.spatial** module, we select the closest node in our tree to the sampled point and generate a nominal path towards it. It is important to note that we check for duplicate nodes with the endpoint of the returned trajectory, such that there are no points located within a 0.1 meter circular region about it. If there are no collisions along the trajectory, we add a new node to our tree and initialize its parents along with other needed information based on the requirements of the RRT planning algorithm.

2.4 Connect Node to Point

We are required to reconnect nodes for the RRT* algorithm in the case of a better cost-to-come connection being available. A path is generated from the starting node to the end node depending on the relative position between the two locations and the robot's current heading. In the case when the nodes are too close or can be reached by driving the robot straight without changing its heading, the function **connect_node_to_point(...)** returns with no results or a linearly interpolated path in 2D as per the logic implemented. Otherwise, a circular arc is generated constrained by the robot's heading, which serves as the tangent to the circle, and the two endpoints of the path to be defined. In order to make the implementation easier, the finish location is mapped into the robots reference frame through pose transformation to simplify the logic for interpolating the robot's heading along the path.

2.5 Trajectory Cost

The cost used in our implementation is that of Euclidean distance, such that for every to consecutive points along the trajectory, a distance is computed according to the distance metric and summed through for the entirely of the path.

2.6 Updating Children

A recursive approach is required for the updating the children of a node if rewiring is performed after finding the optimal connection for a node. We start with the parent node and recursively call **update_children(...)** on its children nodes to update the cost-to-come up to the highest level of the tree (if one visualizes it hierarchically).

2.7 RRT* Local Planning

The RRT* follows the same procedure as the RRT algorithm presented in section 2.3 earlier in our report, with a few extra steps after adding the node to our tree. We find all the nearest neighbours with respect to the new node within a ball radius **rN** as a function of the current number of nodes in the tree, which is computed by through the **ball_radius()** function, and search for a cost optimal connection through the list provided by the **cKDTree** methods - if a better connection is found in terms of cost, the newly added node information is updated. As the RRT* algorithm dictates, optimal connections through the new node are queried and made if possible.

2.8 Local Planning

The functions for trajectory rollout and collision checking were reused, and adapted to support the path following algorithm along with the helper functions they depend on. Trajectories are propagated for a combination of input every iteration and checked for collision in order to filter them out of our options. In the **calculate˙cost** function, we prioritize controls that take us closer to the endpoint with as little difference as possible in input combinations from the previous control by assigning cost variables - we played around with the cost values and selected a combination that worked well for our paths and implementations. It is important to note that different tolerances were selected for Myhal and Willow garage simulations/demo to execute the path efficiently.

3 Paths

3.1 Visuals

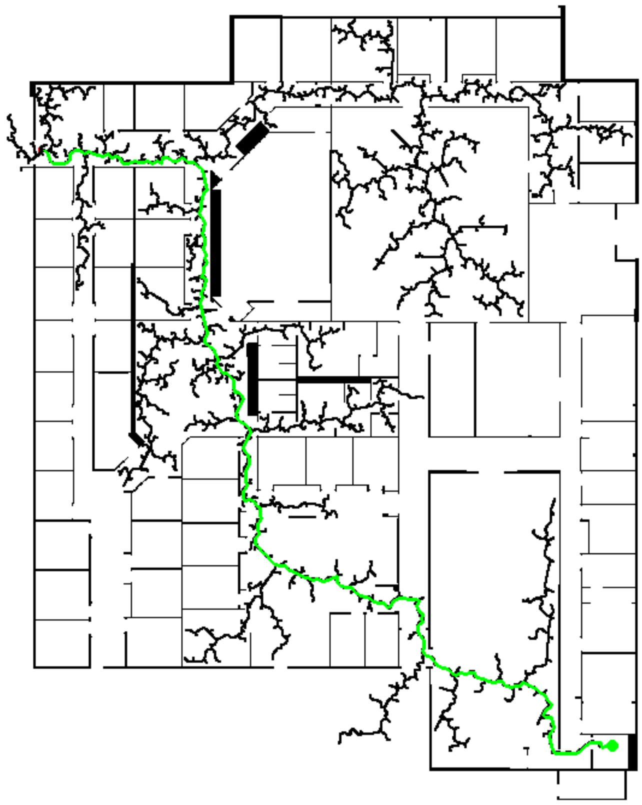

Figure 1: Tree and optimal path generated by RRT implementation.

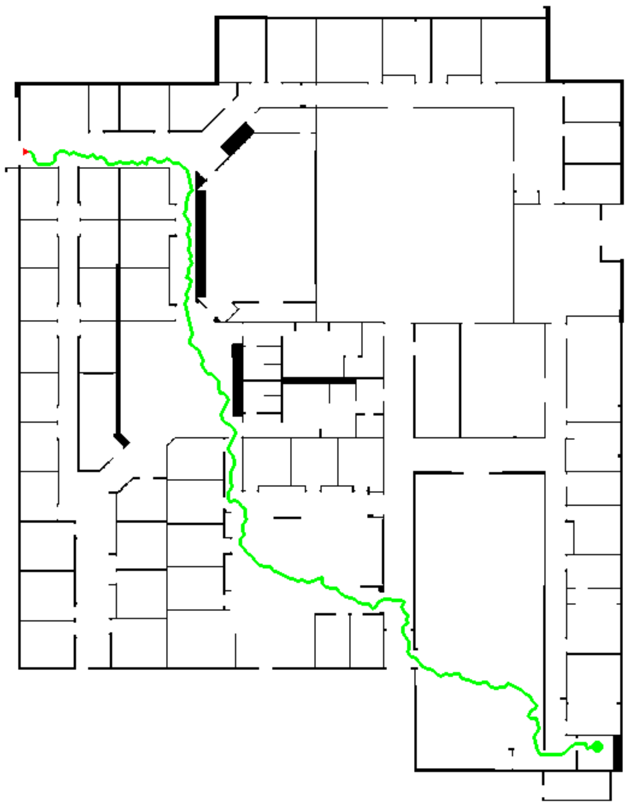

Figure 2: Optimal path generated by RRT implementation.

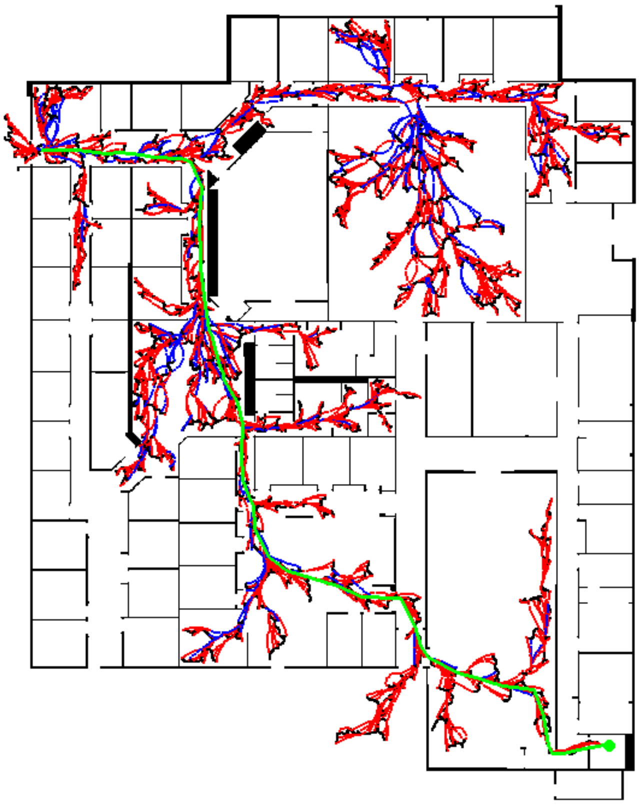

Figure 3: Node rewiring and tree generated by RRT* implementation - red represents first order rewiring of the new nodes and blue indicates last node rewiring implemented in accordance with the skeleton code.

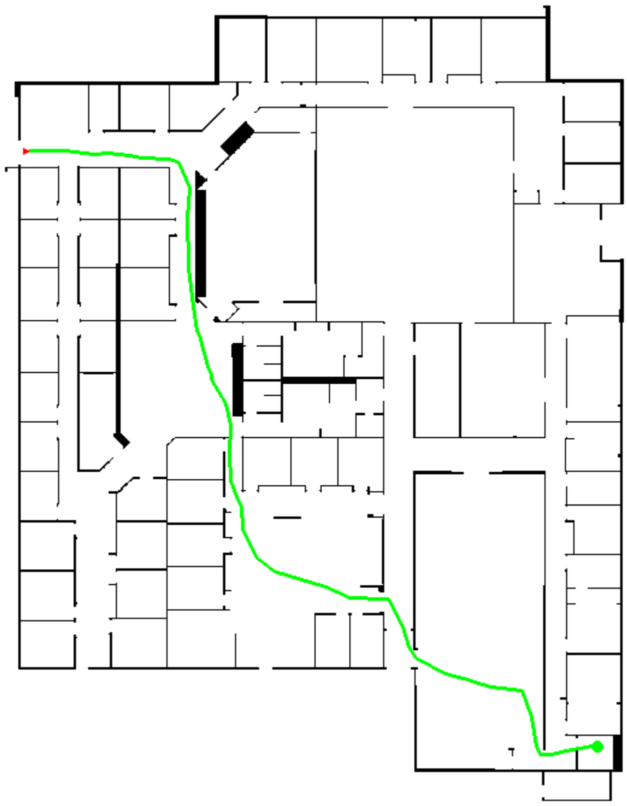

Figure 4: Optimal path generated by RRT* implementation.

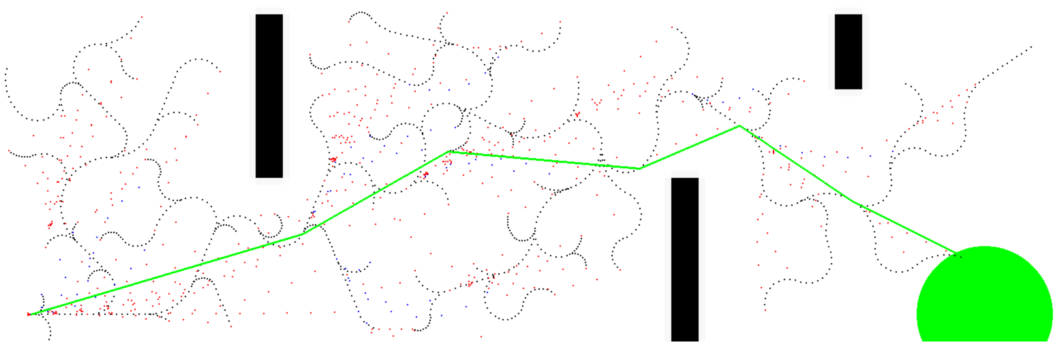

Figure 5: Optimal path generated by RRT* implementation for the Myhal demo.

3.2 Comments

We were able to successfully solve the path planning problem and determine a seed that also provides the shortest path within the Willow garage map. Because of the rewiring using non-holonomic constraints, the trajectory generated by the RRT* is much smoother and more optimal than the RRT path. It is also important to node that because of the rewiring and nature of rolling out trajectories with negative velocities, the orientation of the robot at certain locations can be flipped in the opposite direction of movement - in the future, one can post process the optimal path data to fix the orientation from one way-point to another such that the robot does not circle when reaching a location to match its orientation.

4 Simulation and Demo

A .zip file is included with our submission on Quercus containing the recordings of the Willow garage solution and in-person demo of solving the Myhal maze. We would like to note that VNC has not been working correctly for us in terms of resolution and the videos are a record of our playback in the lab. Moreover, we did not notice at the time that the location of goal was misplaced and the video shows the solution presented below for the Willow garage. Nevertheless, this demonstrates that our path following algorithm works successfully, and illustrates what was mentioned under Section 3.2 in terms of heading alignment of the robot. We noticed that when the following algorithm was ran in Turtlebot3, the robot seemed more stable and transitions from command to command were much smoother.

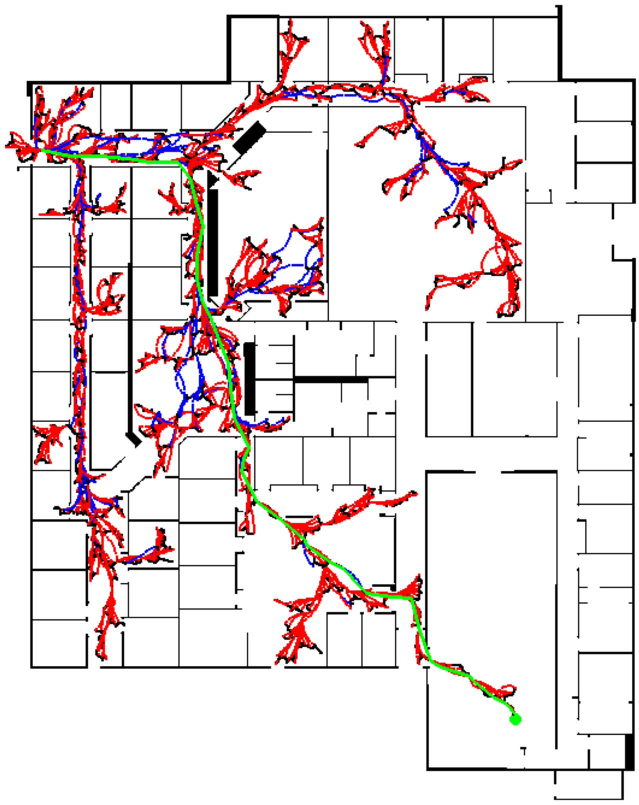

Figure 6: Node rewiring and tree generated by RRT* implementation - red represents first order rewiring of the new nodes and blue indicates last node rewiring implemented in accordance with the skeleton code - path recorded in simulation.

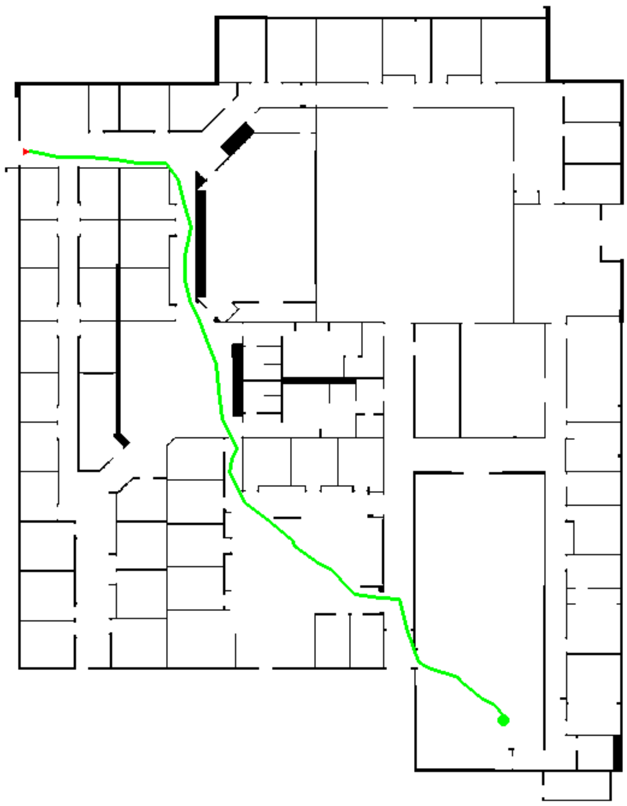

Figure 7: Optimal path generated by RRT* implementation - path recorded in simulation.

5 Source Code

5.1 Planning : l2_planning.py

```
#!/usr/bin/env python3
#Standard Libraries
import sys
import numpy as np
import yaml
import pygame
import time
import pygame_utils
import matplotlib.image as mpimg
from skimage.draw import disk
from scipy.linalg import block_diag
from scipy.spatial import cKDTree
import math
import matplotlib.pyplot as plt

## Myhal SEED = 10 FOR RRT
## Myhal SEED = 16 FOR RRT*
# RRT-STAR and RRT Seed - Willow (24)
np.random.seed(24)

def load_map(filename):
    im = mpimg.imread("./maps/" + filename)
    if len(im.shape) > 2:
        im = im[:,:,0]
    im_np = np.array(im)  #Whitespace is true, black is false
    #im_np = np.logical_not(im_np)
    return im_np

def load_map_yaml(filename):
    with open("./maps/" + filename, "r") as stream:
            map_settings_dict = yaml.safe_load(stream)
    return map_settings_dict

#Node for building a graph
class Node:
    def __init__(self, point, parent_id, cost):
        self.point = point # A 3 by 1 vector [x, y, theta]
        self.parent_id = parent_id # The parent node id that leads to this node (There should only every be one parent in RRT)
        self.trajFromParent = None
        self.cost = cost # The cost to come to this node
        self.children_ids = [] # The children node ids of this node
        return

#Path Planner
class PathPlanner:
    #A path planner capable of perfomring RRT and RRT*
    def __init__(self, map_filename, map_setings_filename, goal_point, stopping_dist):
        #Get map information
        self.occupancy_map = load_map(map_filename)
        self.map_shape = self.occupancy_map.shape
        self.map_settings_dict = load_map_yaml(map_setings_filename)

        #Get the metric bounds of the map
        self.bounds = np.zeros([2,2]) #m
        self.bounds[0, 0] = self.map_settings_dict["origin"][0]
        self.bounds[1, 0] = self.map_settings_dict["origin"][1]
        self.bounds[0, 1] = self.map_settings_dict["origin"][0] + self.map_shape[1] * self.map_settings_dict["resolution"]
        self.bounds[1, 1] = self.map_settings_dict["origin"][1] + self.map_shape[0] * self.map_settings_dict["resolution"]

        # User Defined Variables
        self.cellRes = self.map_settings_dict["resolution"]
        self.originTheta = self.map_settings_dict["origin"][2]
        self.originXY = np.array([[self.map_settings_dict["origin"][0]],
                                  [self.map_settings_dict["origin"][1]]])

```

```
        # map dimensions (bounds the region of the actual labyrinth not the whole 'world')
        self.topL = [-3, 12]
        self.topR = [45, -49]

        #Robot information
        self.robot_radius = 0.22 #m
        ## CHOSEN TO MATCH THE follow_path.py range of inputs during trjectory rollout
        self.vel_max = 0.28 #m/s (Feel free to change!)
        self.rot_vel_max = 1.82 #rad/s (Feel free to change!)

        self.v_options = np.linspace(-self.vel_max, self.vel_max, 5)
        self.w_options = np.linspace(-self.rot_vel_max, self.rot_vel_max, 5)

        self.mapIsMyhal = False

        #Goal Parameters
        self.goal_point = goal_point #m
        self.stopping_dist = stopping_dist #m

        #Trajectory Simulation Parameters
        self.timestep = 1.5 #s
        self.num_substeps = 10

        #Planning storage
        self.nodes = [Node(np.zeros((3,1)), -1, 0)]

        #RRT* Specific Parameters
        self.lebesgue_free = np.sum(self.occupancy_map) * self.map_settings_dict["resolution"]**2
        self.zeta_d = np.pi
        self.gamma_RRT_star = 2*(1 + 1/2)**(1/2)*(self.lebesgue_free / self.zeta_d)**(1/2)
        self.gamma_RRT = self.gamma_RRT_star + .1
        self.epsilon = 2.5

        ## FIX BUG WHEN LOADING THE MAP - pygame_utils.py was modified accordingly
        # Pygame window for visualization
        if map_filename == 'myhal.png':
            shape = (self.occupancy_map.shape[1]*10, self.occupancy_map.shape[0]*10)
            self.rot_vel_max = np.pi/2
            self.v_options = np.linspace(0, self.vel_max, 5)
            self.w_options = np.linspace(-self.rot_vel_max, self.rot_vel_max, 5)
            self.num_substeps = 10
            self.mapIsMyhal = True
        else:
            shape = (900, 900)

        self.window = pygame_utils.PygameWindow(
            "Path Planner", shape, self.occupancy_map.T.shape, self.map_settings_dict, self.goal_point, self.stopping_dist, map_filename)
        return

    #Functions required for RRT
    def sample_map_space(self):
        #Return an [x,y] coordinate to drive the robot towards

        probGoal = np.random.rand()

        if probGoal < 0.05:
            randX = self.goal_point[0][0] + 3*self.stopping_dist*np.random.rand()
            randY = self.goal_point[1][0] + 3*self.stopping_dist*np.random.rand()
            return np.array([[randX], [randY]])

        if not self.mapIsMyhal:
            randX = np.random.rand()*(self.topR[0] - self.topL[0]) + self.topL[0]
            randY = np.random.rand()*(self.topR[1] - self.topL[1]) + self.topL[1]
        else:
            randX = np.random.rand()*(self.bounds[0, 1] - self.bounds[0, 0]) + self.bounds[0, 0]
            randY = np.random.rand()*(self.bounds[1, 1] - self.bounds[1, 0]) + self.bounds[1, 0]

```

```
        return np.array([[randX], [randY]])

    def check_if_duplicate(self, point):
        #Check if point is a duplicate of an already existing node

        closest = self.closest_node(point)
        closestPt = self.nodes[closest].point[:2,:].reshape((2,1))

        if np.linalg.norm(point - closestPt) <= 0.1:
            return True

        return False

    def closest_node(self, point):
        #Returns the index of the closest node

        dist = cKDTree(np.stack([node.point[:-1, :] for node in self.nodes], axis = 0).squeeze(-1))
        bestDist , Id = dist.query(point.T, k = 1)

        return Id[0]

    def simulate_trajectory(self, node_i: Node, point_s):
        #Simulates the non-holonomic motion of the robot.
        #This function drives the robot from node_i towards point_s. This function does have many solutions!
        #node_i is a 3 by 1 vector [x;y;theta] this can be used to construct the SE(2) matrix T_{OI} in course notation
        #point_s is the sampled point vector [x; y]

        robot_traj = None

        v, w = self.robot_controller(node_i.point, point_s)

        trajBest = self.trajectory_rollout(v, w, node_i.point)

        deltaXY = trajBest[:2,:] - point_s

        batchDist = np.linalg.norm(deltaXY, axis = 0)

        minId = np.argmin(batchDist)

        robot_traj = trajBest[:,:(minId + 1)]

        return robot_traj

    def robot_controller(self, poseS, pointE):
        #This controller determines the velocities that will nominally move the robot from node i to node s
        #Max velocities should be enforced

        # initialize output to default values
        v = 0
        w = 0
        # loop variable
        bestDist = np.inf
        # determine control law that takes us closest to end-point pointE
        for i in range(0, len(self.v_options)):

            v_test = self.v_options[i]

            for j in range(0,len(self.w_options)):

                w_test = self.w_options[j]

                traj_ij = self.trajectory_rollout(v_test, w_test, poseS)

                deltaXY = traj_ij[:2,:] - pointE

                batchDist = np.linalg.norm(deltaXY, axis = 0)

                minId = np.argmin(batchDist)

```

```
                if batchDist[minId] < bestDist:
                    v = v_test
                    w = w_test
                    bestDist = batchDist[minId]

        return v, w

    def trajectory_rollout(self, vel, rot_vel, x_0):

        # Given your chosen velocities determine the trajectory of the robot for your given timestep
        # The returned trajectory should be a series of points to check for collisions

        # x_k_1 = x_k + B(x_k)u_k*dt
        B = lambda x_theta : np.array([[np.cos(x_theta), 0],[np.sin(x_theta), 0],[0, 1]])

        # initialize output
        trajectory = x_0

        # prepare variables needed for computation
        dt = self.timestep/self.num_substeps
        u = np.array([[vel],[rot_vel]])
        # control is computed from the current pose
        # in the global reference frame
        currState = x_0
        nextState = np.zeros((3,1))

        for i in range(0, self.num_substeps):

            # compute B(x_k)
            B_k = B(currState[2][0])
            # compute new state
            nextState = currState + np.dot(B_k,u)*dt

            # adjust heading to be in [-pi, pi]
            if nextState[2] > np.pi:
                nextState[2] = nextState[2] - 2*np.pi
            elif nextState[2] < -np.pi:
                nextState[2] = nextState[2] + 2*np.pi

            # save current state and progress to the next timestep
            trajectory = np.hstack((trajectory, nextState))

            currState = nextState

        return trajectory

    def point_to_cell(self, point):

        #Convert a series of [x,y] points in the map to the indices for the corresponding cell in the occupancy map
        #point is a 2 by N matrix of points of interest

        # compute rotation matrix to map points on the reference
        # frame if a rotation is present
        theta = self.originTheta
        R = np.array([[np.cos(theta), np.sin(theta)],
                      [-np.sin(theta), np.cos(theta)]])

        # transform points to cell coordinates
        pts_origin = (np.dot(R, point) - np.dot(R, self.originXY))/self.cellRes

        ptsX = pts_origin[0, :]
        ptsY = self.map_shape[0] - pts_origin[1,:]
        ptsOut = np.vstack((ptsX, ptsY))

        return ptsOut.astype(int) #(2, N)

    def points_to_robot_circle(self, points):

        #Convert a series of [x,y] points to robot map footprints for collision detection
```

```python
        #Hint: The disk function is included to help you with this function

        # transform robot coordinates to cell coordinates
        cellPos = self.point_to_cell(points)

        # create a base region for the radius specified
        radRes = int(self.robot_radius/self.cellRes) + 1

        # attempt to be more conservative with Myhal map
        if self.mapIsMyhal:
            radRes += 1

        rr, cc = disk((0, 0), radRes)
        baseCoords = np.vstack((rr,cc), dtype = np.int32)

        # create result place holder
        tempX = np.zeros((2,1))
        pts2Rob = {}

        for i in range(0, cellPos.shape[1]):

            # check for duplicates
            keyLoc = (cellPos[0][i], cellPos[1][i])

            if keyLoc not in pts2Rob:

                # extract current points
                tempX[0] = cellPos[0][i]
                tempX[1] = cellPos[1][i]
                # map the occupancy region
                tempDiff = tempX + baseCoords # shape of (2, 2*radRes + 1)
                # adjust for corner cases
                tempDiff[0, tempDiff[0] >= self.occupancy_map.shape[1]] = self.occupancy_map.shape[1] - 1
                tempDiff[1, tempDiff[1] >= self.occupancy_map.shape[0]] = self.occupancy_map.shape[0] - 1

                remCells = np.where(tempDiff < 0)[1]
                tempDiffDel = np.delete(tempDiff, remCells, axis = 1)
                # save in dictionary only unique coordinates and entries
                pts2Rob[keyLoc] = np.unique(tempDiffDel, axis = 1)

        return pts2Rob

    def checkCollision(self, points):

        # points are of shape (2, N) in discrete coordinates

        # get the occupancy map for each region
        pts2RobPoses = self.points_to_robot_circle(points) # dictionary with occupancy regions

        # check if pose is collision free
        for key in pts2RobPoses:

            # get occupancy map: (2, M) array
            mapOcc = pts2RobPoses[key].astype(int)

            # check if any of the cells contains a value of 0 and report collision as True
            crash = np.any(self.occupancy_map[mapOcc[1,:], mapOcc[0, :]] == 0)

            if crash:
                return True

        return False

    #Note: If you have correctly completed all previous functions, then you should be able to create a working RRT function

    #RRT* specific functions
    def ball_radius(self):
        #Close neighbor distance
```

```
        card_V = len(self.nodes)
        return min(self.gamma_RRT * (np.log(card_V) / card_V ) ** (1.0/2.0), self.epsilon)

    def connect_node_to_point(self, node_i, point_f):
        #Given two nodes find the non-holonomic path that connects them
        #Settings
        #node is a 3 by 1 node
        #point is a 2 by 1 point

        # initialize return variable
        resTraj = None
        tooClose = False

        # partition input node
        point_s = node_i[:2,0].reshape(2,1)
        robHead = node_i[2,0]

        # important info : distance and bearing angle to the robots refernce frame
        dist = np.linalg.norm(point_s - point_f)
        theta = np.arctan2(point_f[1,0] - point_s[1,0], point_f[0,0] - point_s[0,0])

        if dist <= 0.1:

            tooClose = True

        elif (abs(theta - robHead) < 1e-3) or ((abs(abs(theta - robHead) - np.pi) < 1e-3)):

            # choose an appropriate step resolution according to the distance
            stepNum = 10
            resTraj = np.linspace(point_s, point_f, num = stepNum)[:,:,0].T
            resTraj = np.vstack((resTraj, robHead*np.ones((1, stepNum))))

        else:

            # tranform point_f to robots frame through T_vw
            R_vw = np.array([[np.cos(robHead), np.sin(robHead)],
                             [-np.sin(robHead), np.cos(robHead)]])
            o_vw = - np.dot(R_vw, point_s)

            T_vw = np.hstack((R_vw, o_vw))
            T_vw = np.vstack((T_vw, np.array([[0,0,1]])))
            T_inv = np.linalg.inv(T_vw)

            # augment point_f
            point_f_w = np.vstack((point_f, np.array([[1]])))
            point_f_v = np.dot(T_vw, point_f_w)

            # compute the the center of circle by using vector S->F
            alpha = point_f_v[0,0]/point_f_v[1,0]
            xC = 0
            yC = alpha*point_f_v[0,0]/2 + point_f_v[1,0]/2
            # radius of circle
            r = abs(yC)

            # compute arc-angle
            cosRho = 1 - 2*((dist/(2*r))**2)

            # ILL-CONDITIONED PROBLEM - SKIP CONNECTION
            if abs(abs(cosRho) - 1) < 1e-6:
                return (True, None)

            rho = np.arctan2(np.sqrt(1-cosRho**2), cosRho)

            # compute arc-length
            l = rho*r
            # select an appropriate stepsize
            stepNum = np.ceil(l/self.robot_radius).astype(np.int32) + 1
            # determine direction of traversing the arc length for
            # each of the possible combinations (4 to consider)

            dirY = -1 # decreasing with each substep
```

```
            dirX = -1 # decreasing with each substep

            if yC > 0:
                dirY = 1

            if point_f_v[0,0] > 0:
                dirX = 1

            # start determining the trajectory
            dRho = rho/stepNum
            resTraj = node_i
            rhoAcc = 0

            ## FOR DEBUGGING
            # plt.figure()
            # plt.plot(0, 0, 'rX')
            # plt.plot(point_f_v[0,0], point_f_v[1,0], 'ro')

            for i in range(0, stepNum + 1):

                # compute next waypoint in vehicle frame
                currPt = np.array([[r*np.sin(rhoAcc)*dirX + xC],
                                   [-r*np.cos(rhoAcc)*dirY + yC],
                                   [1]])

                ## FOR DEBUGGING
                # plt.plot(currPt[0,0], currPt[1,0], 'b.')

                # transform to world frame and update heading
                tempPt = np.dot(T_inv, currPt)
                rhoAcc += dRho
                tempPt[2, 0] = robHead + rhoAcc*(dirX*dirY)

                if tempPt[2] > np.pi:
                    tempPt[2] = tempPt[2] - 2*np.pi
                elif tempPt[2] < -np.pi:
                    tempPt[2] = tempPt[2] + 2*np.pi

                # save result
                resTraj = np.hstack((resTraj, tempPt))

            ## FOR DEBUGGING
            # plt.close()


        ## FOR DEBUGGING
        # if not tooClose:
        #     plt.figure()
        #     plt.plot(node_i[0,0], node_i[1,0], 'rX')
        #     plt.plot(point_f[0,0], point_f[1,0], 'ro')

        #     for j in range(0,resTraj.shape[1]):
        #         plt.plot(resTraj[0,j], resTraj[1,j], 'b.')
        #     plt.close()

        return (tooClose, resTraj)

    def cost_to_come(self, trajectory_o):
        #The cost to get to a node from lavalle (2 x N)

        # compute total euclidean distance along the path
        result = 0

        for i in range(1, trajectory_o.shape[1]):
            result += np.linalg.norm(trajectory_o[:2, i].reshape((2,1)) - trajectory_o[:2, i-1].reshape((2,1)))

        return result

    def update_children(self, node_id):
        #Given a node_id with a changed cost, update all connected nodes with the new cost

```

```
        parent = self.nodes[node_id]

        # iterate through the children recursively and update the cost
        for childID in parent.children_ids:
            trajParent = self.nodes[childID].trajFromParent
            self.nodes[childID].cost = parent.cost + self.cost_to_come(trajParent)
            self.update_children(childID)

        return

    #Planner Functions
    def rrt_planning(self):
        #This function performs RRT on the given map and robot
        #You do not need to demonstrate this function to the TAs, but it is left in for
    you to check your work

        i = 0

        while True: #Most likely need more iterations than this to complete the map!

            #Sample map space
            point = self.sample_map_space()

            #Get the closest point
            closest_node_id = self.closest_node(point)

            # Simulate driving the robot towards the closest point
            trajectory_o = self.simulate_trajectory(self.nodes[closest_node_id], point)

            # check how close we were able to reach our goal (we may not be able
            # to reach the point exactly or at all)
            lastPt = trajectory_o[:,-1].reshape((3,1))
            lastPt2D = lastPt[:2,0].reshape((2,1))

            # check if a similar point is already in our tree
            checkDuplicate = self.check_if_duplicate(lastPt2D)

            if checkDuplicate:
                continue

            didCollide = self.checkCollision(trajectory_o[:2,:])

            if not didCollide:

                # add new node on our list
                newNode = Node(lastPt, closest_node_id, 0)
                newNode.trajFromParent = trajectory_o
                self.nodes.append(newNode)
                self.nodes[closest_node_id].children_ids.append(len(self.nodes) - 1)

                # check if we are at goal
                if np.linalg.norm(lastPt2D - self.goal_point) < self.stopping_dist:
                    break

                # # FOR DEBUGGING
                # for j in range(0, trajectory_o.shape[1]):
                #    self.window.add_point(trajectory_o[:-1, j].copy())

            i += 1

        return self.nodes

    def rrt_star_planning(self):
        #This function performs RRT* for the given map and robot
        while True: #Most likely need more iterations than this to complete the map!
            #Sample map space
            point = self.sample_map_space()

            #Get the closest point
            closest_node_id = self.closest_node(point)

            # Simulate driving the robot towards the closest point
            trajectory_o = self.simulate_trajectory(self.nodes[closest_node_id], point)
```

```

        # check how close we were able to reach our goal (we may not be able
        # to reach the point exactly or at all)
        lastPt = trajectory_o[:,-1].reshape((3,1))
        lastPt2D = lastPt[:2,0].reshape((2,1))

        # check if a similar point is already in our tree
        checkDuplicate = self.check_if_duplicate(lastPt2D)

        if checkDuplicate:
            continue

        didCollide = self.checkCollision(trajectory_o[:2,:])

        if not didCollide:

            # add new node on our list
            newNode = Node(lastPt, closest_node_id, 0)
            newNode.trajFromParent = trajectory_o
            newNode.cost = self.cost_to_come(trajectory_o) + self.nodes[closest_node_id].cost
            self.nodes.append(newNode)
            self.nodes[closest_node_id].children_ids.append(len(self.nodes) - 1)

            # compute radius of neighbours
            rN = self.ball_radius()

            # find nearest neighbours within rN
            dist = cKDTree(np.stack([node.point[:-1, :] for node in self.nodes], axis = 0).squeeze(-1))
            Id = dist.query_ball_point(lastPt[:2,0], r = rN)

            currCost = newNode.cost
            newId = len(self.nodes) - 1
            currParentId = closest_node_id
            currTraj = trajectory_o

            ## FOR DEBUGGING
            # for j in range(0, trajectory_o.shape[1]):
            #     self.window.add_point(trajectory_o[:-1, j].copy(), color = (0,0,0))

            lowerCost = False

            #Last node rewire
            for n in Id:

                closeFlag, tempTraj = self.connect_node_to_point(self.nodes[n].point, lastPt2D)

                if (n == newId) or closeFlag:
                    continue

                # check for collision
                didCollideTemp = self.checkCollision(tempTraj[:2,:])

                if not didCollideTemp:

                    # compute cost
                    tempCost = self.cost_to_come(tempTraj) + self.nodes[n].cost

                    if currCost >= tempCost:
                        currCost = tempCost
                        currTraj = tempTraj
                        currParentId = n
                        #lowerCost = True


            # rewire child
            if currParentId != closest_node_id or lowerCost:
                # remove child from the list
                self.nodes[closest_node_id].children_ids.remove(newId)
                # update new parent data
```

```
                    self.nodes[newId].trajFromParent = currTraj
                    self.nodes[newId].cost = currCost
                    self.nodes[newId].parent_id = currParentId
                    self.nodes[currParentId].children_ids.append(newId)

                    ## FOR DEBUGGING
                    # for j in range(0, currTraj.shape[1]):
                    #     self.window.add_point(currTraj[:-1, j].copy(), color = (255,0,0))

                # Close node rewire
                for n in Id:

                    # temporary variable
                    tempPId = self.nodes[n].parent_id

                    closeFlag, tempTraj = self.connect_node_to_point(lastPt, self.nodes[n].point[:2,0].reshape((2,1)))

                    if closeFlag or (n == newId):
                        continue

                    # check for collision
                    didCollideTemp = self.checkCollision(tempTraj[:2,:])

                    if not didCollideTemp:

                        # compute cost and rewire if possible
                        tempCost = self.cost_to_come(tempTraj) + self.nodes[newId].cost

                        if tempCost <= self.nodes[n].cost:
                            self.nodes[tempPId].children_ids.remove(n)
                            self.nodes[n].parent_id = newId
                            self.nodes[n].cost = tempCost
                            self.nodes[n].trajFromParent = tempTraj
                            self.nodes[newId].children_ids.append(n)
                            self.update_children(n)

                            ## FOR DEBUGGING
                            # for j in range(0, currTraj.shape[1]):
                            #     self.window.add_point(currTraj[:-1, j].copy(), color = (0,0,255))


                # check if we are at goal
                if np.linalg.norm(lastPt2D - self.goal_point) < self.stopping_dist:
                    break

        return self.nodes

    def recover_path(self, node_id = -1):
        path = [self.nodes[node_id].point]
        current_node_id = self.nodes[node_id].parent_id
        while current_node_id > -1:
            path.append(self.nodes[current_node_id].point)
            current_node_id = self.nodes[current_node_id].parent_id
        path.reverse()
        return path

def main():

    #Set map information
    # map_filename = "myhal.png"
    # map_setings_filename = "myhal.yaml"
    # #robot information
    # goal_point = np.array([[7], [0]]) #m

    #Set map information
    map_filename = "willowgarageworld_05res.png"
    map_setings_filename = "willowgarageworld_05res.yaml"

    #robot information
    goal_point = np.array([[42], [-44]]) #m
```

```
    stopping_dist = 0.5 #m

    # HAD SOME ISSUES IN MY VM SO I INCREASED THE LIMIT
    sys.setrecursionlimit(4000)
    print("Recursion limit set to {}".format(sys.getrecursionlimit()))

    #RRT precursor
    path_planner = PathPlanner(map_filename, map_setings_filename, goal_point,
    stopping_dist)
    nodes = path_planner.rrt_planning()
    node_path_metric = np.hstack(path_planner.recover_path())

    for i in range (1, node_path_metric.shape[1]):
        pt1 = node_path_metric[:, i-1].reshape((3,1))
        pt2 = node_path_metric[:, i].reshape((3,1))
        path_planner.window.add_line(pt1[:2, 0].copy(), pt2[:2, 0].copy(), width = 3,
    color = (0, 255, 0))

    pygame.image.save(path_planner.window.screen, f"image.png")

    #Leftover test functions
    np.save("shortest_path.npy", node_path_metric)

if __name__ == '__main__':
    main()
```

5.2 Navigation : l2_follow_path.py

```
#!/usr/bin/env python3
from __future__ import division, print_function
import os

import numpy as np
from scipy.linalg import block_diag
from skimage.draw import disk
from scipy.spatial.distance import cityblock
import rospy
import tf2_ros
import matplotlib.pyplot as plt

# msgs
from geometry_msgs.msg import TransformStamped, Twist, PoseStamped
from nav_msgs.msg import Path, Odometry, OccupancyGrid
from visualization_msgs.msg import Marker

# ros and se2 conversion utils
import utils


TRANS_GOAL_TOL = .25  # m, tolerance to consider a goal complete
ROT_GOAL_TOL = .6 #.6  # rad, tolerance to consider a goal complete
TRANS_VEL_OPTS = [0, 0.025, 0.13, 0.26]  # m/s, max of real robot is .26
ROT_VEL_OPTS = np.linspace(-1.82, 1.82, 11)  # rad/s, max of real robot is 1.82
CONTROL_RATE = 5  # Hz, how frequently control signals are sent
CONTROL_HORIZON = 5  # seconds. if this is set too high and INTEGRATION_DT is too low,
    code will take a long time to run!
# INTEGRATION_DT = .025  # s, delta t to propagate trajectories forward by
INTEGRATION_DT = .05
COLLISION_RADIUS = 0.225  # m, radius from base_link to use for collisions, min of
    0.2077 based on dimensions of .281 x .306
ROT_DIST_MULT = .1  # multiplier to change effect of rotational distance in choosing
    correct control
OBS_DIST_MULT = .1  # multiplier to change the effect of low distance to obstacles on a
    path
MIN_TRANS_DIST_TO_USE_ROT = TRANS_GOAL_TOL  # m, robot has to be within this distance to
     use rot distance in cost
PATH_NAME = 'path.npy'  # saved path from l2_planning.py, should be in the same
    directory as this file

# here are some hardcoded paths to use if you want to develop l2_planning and this file
    in parallel
# TEMP_HARDCODE_PATH = [[2, 0, 0], [2.75, -1, -np.pi/2], [2.75, -4, -np.pi/2], [2, -4.4,
```

```
    np.pi]]  # almost collision-free
TEMP_HARDCODE_PATH = [[2, -.5, 0], [2.4, -1, -np.pi/2], [2.45, -3.5, -np.pi/2], [1.5, -4.4, np.pi]]  # some possible collisions


class PathFollower():
    def __init__(self):
        # time full path
        self.path_follow_start_time = rospy.Time.now()

        # use tf2 buffer to access transforms between existing frames in tf tree
        self.tf_buffer = tf2_ros.Buffer()
        self.listener = tf2_ros.TransformListener(self.tf_buffer)
        rospy.sleep(1.0)  # time to get buffer running

        # constant transforms
        self.map_odom_tf = self.tf_buffer.lookup_transform('map', 'odom', rospy.Time(0), rospy.Duration(2.0)).transform
        # print("map odom tf:")
        # print(self.map_odom_tf)

        # subscribers and publishers
        self.cmd_pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
        self.global_path_pub = rospy.Publisher('~global_path', Path, queue_size=1, latch=True)
        self.local_path_pub = rospy.Publisher('~local_path', Path, queue_size=1)
        self.collision_marker_pub = rospy.Publisher('~collision_marker', Marker, queue_size=1)

        # map
        map = rospy.wait_for_message('/map', OccupancyGrid)
        self.map_np = np.array(map.data).reshape(map.info.height, map.info.width)
        self.map_resolution = round(map.info.resolution, 5)
        self.map_origin = -utils.se2_pose_from_pose(map.info.origin)  # negative because of weird way origin is stored
        # print("map origin and map")
        # print(self.map_origin)
        self.map_nonzero_idxes = np.argwhere(self.map_np) #prob use this for colision detection
        # print(map.info)
        # print(np.shape(self.map_np))

        # for debug
        self.map_xy = np.nonzero(self.map_np)
        # print(map_xy)
        # plt.scatter(map_xy[0], map_xy[1])
        # plt.show()
        # exit()


        # collisions
        self.collision_radius_pix = COLLISION_RADIUS / self.map_resolution
        self.collision_marker = Marker()
        self.collision_marker.header.frame_id = '/map'
        self.collision_marker.ns = '/collision_radius'
        self.collision_marker.id = 0
        self.collision_marker.type = Marker.CYLINDER
        self.collision_marker.action = Marker.ADD
        self.collision_marker.scale.x = COLLISION_RADIUS * 2
        self.collision_marker.scale.y = COLLISION_RADIUS * 2
        self.collision_marker.scale.z = 1.0
        self.collision_marker.color.g = 1.0
        self.collision_marker.color.a = 0.5

        # transforms
        self.map_baselink_tf = self.tf_buffer.lookup_transform('map', 'base_link', rospy.Time(0), rospy.Duration(2.0))
        self.pose_in_map_np = np.zeros(3)
        self.pos_in_map_pix = np.zeros(2)
        self.update_pose()

        # path variables
        cur_dir = os.path.dirname(os.path.realpath(__file__))
```

```python

        # to use the temp hardcoded paths above, switch the comment on the following two lines
        self.path_tuples = np.load(os.path.join(cur_dir, 'RRT_star_willow_seed_7.npy')).T
        #self.path_tuples = np.array(TEMP_HARDCODE_PATH)

        self.path = utils.se2_pose_list_to_path(self.path_tuples, 'map')
        self.global_path_pub.publish(self.path)

        # goal
        self.cur_goal = np.array(self.path_tuples[0])
        self.cur_path_index = 0

        # trajectory rollout tools
        # self.all_opts is a Nx2 array with all N possible combinations of the t and v vels, scaled by integration dt
        self.all_opts = np.array(np.meshgrid(TRANS_VEL_OPTS, ROT_VEL_OPTS)).T.reshape(-1, 2)

        # if there is a [0, 0] option, remove it
        all_zeros_index = (np.abs(self.all_opts) < [0.001, 0.001]).all(axis=1).nonzero()[0]
        if all_zeros_index.size > 0:
            self.all_opts = np.delete(self.all_opts, all_zeros_index, axis=0)
        self.all_opts_scaled = self.all_opts * INTEGRATION_DT

        self.num_opts = self.all_opts_scaled.shape[0]
        self.horizon_timesteps = int(np.ceil(CONTROL_HORIZON / INTEGRATION_DT))

        self.rate = rospy.Rate(CONTROL_RATE)
        self.prev_ctrl = np.array([0,0])

        rospy.on_shutdown(self.stop_robot_on_shutdown)
        self.follow_path()

    def trajectory_rollout(self, vel, rot_vel, x_0):

        # Given your chosen velocities determine the trajectory of the robot for your given timestep
        # The returned trajectory should be a series of points to check for collisions

        # x_0 is expected as x, y, theta
        # x_k_1 = x_k + B(x_k)u_k*dt
        B = lambda x_theta : np.array([[np.cos(x_theta), 0],[np.sin(x_theta), 0],[0, 1]])

        # initialize output
        trajectory = np.zeros((3, 1))

        # prepare variables needed for computation
        u = np.array([[vel],[rot_vel]])
        dt = INTEGRATION_DT
        # control is computed from the current pose
        # in the global reference frame
        currState = np.reshape(x_0, (3,1))
        nextState = np.zeros((3,1))

        for i in range(0, self.horizon_timesteps):

            # compute B(x_k)
            B_k = B(currState[2][0])
            # compute new state
            #print(np.shape(B_k))
            #print(np.shape(u))
            nextState = currState + np.dot(B_k,u)*dt

            # print("state propagation")
            # print(currState)
            # print(nextState)

            # adjust heading to be in [-pi, pi]
            if nextState[2] > np.pi:
```

```
                nextState[2] = nextState[2] - 2*np.pi
            elif nextState[2] < -np.pi:
                nextState[2] = nextState[2] + 2*np.pi

            # save current state and progress to the next timestep
            if i == 0:
                trajectory = nextState
            else:
                trajectory = np.hstack((trajectory, nextState))

            currState = nextState

        return np.transpose(trajectory)

    def points_to_robot_circle(self, points):
        #Convert a series of [x,y] points to robot map footprints for collision detection
        #Hint: The disk function is included to help you with this function

        # transform robot coordinates to cell coordinates
        # cellPos = self.point_to_cell(points)
        cellPos = points

        # create a base region for the radius specified
        radRes = int(COLLISION_RADIUS/self.map_resolution) # TODO: check if self map resolution is legit
        rr, cc = disk((0, 0), radRes)
        baseCoords = np.vstack((rr,cc), dtype = np.int32)

        # create result place holder
        tempX = np.zeros((2,1))
        pts2Rob = {}

        for i in range(0, cellPos.shape[1]):

            # check for duplicates
            keyLoc = (cellPos[0][i], cellPos[1][i])

            if keyLoc not in pts2Rob:

                # extract current points
                tempX[0] = cellPos[0][i]
                tempX[1] = cellPos[1][i]
                # map the occupancy region
                tempDiff = tempX + baseCoords # shape of (2, 2*radRes + 1)
                # adjust for corner cases
                tempDiff[0, tempDiff[0] >= self.map_np.shape[1]] = self.map_np.shape[1] - 1
                tempDiff[1, tempDiff[1] >= self.map_np.shape[0]] = self.map_np.shape[0] - 1
                remCells = np.where(tempDiff < 0)[1]
                tempDiffDel = np.delete(tempDiff, remCells, axis = 1)
                # save in dictionary only unique coordinates and entries
                pts2Rob[keyLoc] = np.unique(tempDiffDel, axis = 1)

        return pts2Rob

    def checkCollision(self, points_T):
        # points are of shape (2, N) in discrete coordinates
        points = np.transpose(points_T)
        # get the robot occupancy map for each set of center points
        pts2RobPoses = self.points_to_robot_circle(points) # dictionary with occupancy regions
        # dictionary with keys being center points, and data being an array of coordinates of rob occupancy

        # check if pose is collision free
        for key in pts2RobPoses:

            # get robot occupancy map: (2, M) array
            mapOcc = pts2RobPoses[key].astype(int) # coordiantes based on set passed in by "points"
            # print("occupied cells by key:{}".format(key))
```

```python
            # print(mapOcc)

            # plt.scatter(-self.map_xy[0], self.map_xy[1])
            # plt.scatter(-mapOcc[1,:], mapOcc[0,:])
            # plt.scatter(-key[1], key[0], s = 100)
            # plt.scatter(-points[1,0], points[0,0])
            # plt.show()
            # exit()

            # check if any of the environment occupancy map cells occupied by the robot contains a value of 0 and report collision as True
            crash = np.any(self.map_np[mapOcc[1, :], mapOcc[0, :]] == 100) # TODO: ensure this is how map_np works
            if crash:
                return True

        return False

    def calculate_cost(self, cand_opt, end_pt):
        # 2 preferences: low dist from goal, low change from previous ctrl
        trans_factor = 16
        rot_factor = 0
        diff_rot_factor = 0

        # difference in goal poses penalized
        curr_goal = self.cur_goal
        pose_diff = np.abs(curr_goal - end_pt)
        loc_cost = trans_factor*(pose_diff[0] + pose_diff[1])
        rot_cost = rot_factor*pose_diff[2] / loc_cost

        # change in control penalzie
        curr_trans_opt = cand_opt[0]
        last_trans_opt = self.prev_ctrl[0]
        curr_rot_opt = cand_opt[1]
        last_rot_opt = self.prev_ctrl[1]
        ctrl_chg_cost = np.abs(curr_trans_opt - last_trans_opt) + diff_rot_factor*np.abs(curr_rot_opt - last_rot_opt)

        return loc_cost + rot_cost + ctrl_chg_cost # TODO: maybe implement a saturating cost for rotational error

    def follow_path(self):
        while not rospy.is_shutdown():
            # timing for debugging...loop time should be less than 1/CONTROL_RATE
            tic = rospy.Time.now()

            self.update_pose()
            self.check_and_update_goal()

            t1 = rospy.Time.now()
            # start trajectory rollout algorithm
            local_paths = np.zeros([self.horizon_timesteps + 1, self.num_opts, 3])
            local_paths[0] = np.atleast_2d(self.pose_in_map_np).repeat(self.num_opts, axis=0)

            # propogate trajectory forward, assuming perfect control of velocity and no dynamic effects

            for i, opts in enumerate(self.all_opts):
                trans_vel = opts[0]
                rot_vel = opts[1]
                pred_traj = self.trajectory_rollout(trans_vel, rot_vel, self.pose_in_map_np)
                local_paths[1:,i,:] = pred_traj #assign complete trajectory one at a time
                # print(local_paths[:,0,:])
            # print("start")
            # print(local_paths[0,:,:])
            # print("end pts")
            # print(local_paths[-1,:,:])

            t2 = rospy.Time.now()
            traj_rollout_time = t2 - t1
```

```
        # check all trajectory points for collisions
        # first find the closest collision point in the map to each local path point
        local_paths_pixels = np.rint((self.map_origin[:2] + local_paths[:, :, :2]) / self.map_resolution).astype(int) #round to nearest int as index
        # print("local path shape")
        # print(np.shape(local_paths_pixels))
        # plt.scatter(-self.map_xy[0], self.map_xy[1])
        # plt.scatter(-local_paths_pixels[0,:,1], local_paths_pixels[0,:,0])
        # print(-local_paths_pixels[0,0,0:2])
        #
        # print(-local_paths_pixels[-1,:,0:2])
        # plt.scatter(-local_paths_pixels[-1,:,1], local_paths_pixels[-1,:,0])
        # plt.show()


        valid_opts = list(range(self.num_opts))
        invalid_opts = []
        local_paths_lowest_collision_dist = np.ones(self.num_opts) * 50

        print("DONE: Check the points in local_path_pixels for collisions")
        for opt in range(local_paths_pixels.shape[1]):
            # for timestep in range(local_paths_pixels.shape[0]):

            rob_center_pixels = local_paths_pixels[:,opt,:]

            # plt.scatter(-self.map_xy[0], self.map_xy[1])
            # plt.scatter(-rob_center_pixels[:,1], rob_center_pixels[:,0])
            # plt.show()
            # print(rob_center_pixels)
            # plt.scatter(-self.map_xy[0], self.map_xy[1])
            # plt.scatter(-rob_center_pixels[:,1], rob_center_pixels[:,0])
            # plt.show()
            # print(np.shape(rob_center_pixels))

            if self.checkCollision(rob_center_pixels): # if there is a collision anywhere along timstep
                # print(rob_center_pixels)
                # plt.scatter(-self.map_xy[0], self.map_xy[1])
                # plt.scatter(-rob_center_pixels[:,1], rob_center_pixels[:,0], c='red')
                # plt.show()
                # exit()
                valid_opts.remove(opt) # we immediately remove the option from the lists
        t3 = rospy.Time.now()
        collison_det_time = t3 - t2
        # remove trajectories that were deemed to have collisions
        print("DONE: Remove trajectories with collisions!")

        # calculate final cost and choose best option
        print("DONE: Calculate the final cost and choose the best control option!")
        # final_cost = np.zeros(self.num_opts)
        final_cost = np.zeros(len(valid_opts))
        for i in range(0, len(valid_opts)):
            # print("checking opts")
            # print(local_paths[-1,i,:])
            cur_opt = self.all_opts_scaled[i]
            final_cost[i] = self.calculate_cost(cur_opt, local_paths[-1,valid_opts[i],:])

        if final_cost.size == 0:  # hardcoded recovery if all options have collision
        # if np.count_nonzero(final_cost) == 0:
            control = [-.1, 0]
        else:
            best_opt = valid_opts[final_cost.argmin()]
            # print("chosen_control")
            # print(final_cost.argmin())
            # print(best_opt)
            # print(self.all_opts[best_opt])
            # print(local_paths[-1,best_opt,:])
            control = self.all_opts[best_opt]
            self.local_path_pub.publish(utils.se2_pose_list_to_path(local_paths[:,
```

```
    best_opt], 'map'))

            t4 = rospy.Time.now()
            cost_calc_time = t4 - t3
            # send command to robot
            self.cmd_pub.publish(utils.unicycle_vel_to_twist(control))

            # uncomment out for debugging if necessary
            # print("Selected control: {control}, Loop time: {time}, Max time: {max_time}".format(
            #       control=control, time=(rospy.Time.now() - tic).to_sec(), max_time=1/CONTROL_RATE))
            # print("traj rollout time: {}, collision det time: {}, cost calc time: {}".format(traj_rollout_time, collison_det_time, cost_calc_time))
            self.rate.sleep()

    def update_pose(self):
        # Update numpy poses with current pose using the tf_buffer
        self.map_baselink_tf = self.tf_buffer.lookup_transform('map', 'base_link', rospy.Time(0)).transform
        self.pose_in_map_np[:] = [self.map_baselink_tf.translation.x, self.map_baselink_tf.translation.y,
                                  utils.euler_from_ros_quat(self.map_baselink_tf.rotation)[2]]
        self.pos_in_map_pix = (self.map_origin[:2] + self.pose_in_map_np[:2]) / self.map_resolution
        self.collision_marker.header.stamp = rospy.Time.now()
        self.collision_marker.pose = utils.pose_from_se2_pose(self.pose_in_map_np)
        self.collision_marker_pub.publish(self.collision_marker)

    def check_and_update_goal(self):
        # iterate the goal if necessary
        dist_from_goal = np.linalg.norm(self.pose_in_map_np[:2] - self.cur_goal[:2])
        abs_angle_diff = np.abs(self.pose_in_map_np[2] - self.cur_goal[2])
        rot_dist_from_goal = min(np.pi * 2 - abs_angle_diff, abs_angle_diff)
        if dist_from_goal < TRANS_GOAL_TOL and rot_dist_from_goal < ROT_GOAL_TOL:
            rospy.loginfo("Goal {goal} at {pose} complete.".format(
                    goal=self.cur_path_index, pose=self.cur_goal))
            if self.cur_path_index == len(self.path_tuples) - 1:
                rospy.loginfo("Full path complete in {time}s! Path Follower node shutting down.".format(
                    time=(rospy.Time.now() - self.path_follow_start_time).to_sec()))
                rospy.signal_shutdown("Full path complete! Path Follower node shutting down.")
            else:
                self.cur_path_index += 1
                self.cur_goal = np.array(self.path_tuples[self.cur_path_index])
        else:
            rospy.logdebug("Goal {goal} at {pose}, trans error: {t_err}, rot error: {r_err}.".format(
                goal=self.cur_path_index, pose=self.cur_goal, t_err=dist_from_goal, r_err=rot_dist_from_goal
            ))

    def stop_robot_on_shutdown(self):
        self.cmd_pub.publish(Twist())
        rospy.loginfo("Published zero vel on shutdown.")


if __name__ == '__main__':
    try:
        rospy.init_node('path_follower', log_level=rospy.DEBUG)
        pf = PathFollower()
    except rospy.ROSInterruptException:
        pass
```

5.3 Navigation : l2_follow_path_myhal.py

```
#!/usr/bin/env python3
from __future__ import division, print_function
import os

import numpy as np
```

```python
from scipy.linalg import block_diag
from scipy.spatial.distance import cityblock
from skimage.draw import disk
import rospy
import tf2_ros

# msgs
from geometry_msgs.msg import TransformStamped, Twist, PoseStamped
from nav_msgs.msg import Path, Odometry, OccupancyGrid
from visualization_msgs.msg import Marker

# ros and se2 conversion utils
import utils


TRANS_GOAL_TOL = .1  # m, tolerance to consider a goal complete
ROT_GOAL_TOL = .3  # rad, tolerance to consider a goal complete
TRANS_VEL_OPTS = [0, 0.025, 0.13, 0.26]  # m/s, max of real robot is .26
ROT_VEL_OPTS = np.linspace(-1.82, 1.82, 11)  # rad/s, max of real robot is 1.82
CONTROL_RATE = 5  # Hz, how frequently control signals are sent
CONTROL_HORIZON = 5  # seconds. if this is set too high and INTEGRATION_DT is too low, code will take a long time to run!
INTEGRATION_DT = .025  # s, delta t to propagate trajectories forward by
COLLISION_RADIUS = 0.225  # m, radius from base_link to use for collisions, min of 0.2077 based on dimensions of .281 x .306
ROT_DIST_MULT = .1  # multiplier to change effect of rotational distance in choosing correct control
OBS_DIST_MULT = .1  # multiplier to change the effect of low distance to obstacles on a path
MIN_TRANS_DIST_TO_USE_ROT = TRANS_GOAL_TOL  # m, robot has to be within this distance to use rot distance in cost
PATH_NAME = 'path.npy'  # saved path from l2_planning.py, should be in the same directory as this file

# here are some hardcoded paths to use if you want to develop l2_planning and this file in parallel
# TEMP_HARDCODE_PATH = [[2, 0, 0], [2.75, -1, -np.pi/2], [2.75, -4, -np.pi/2], [2, -4.4, np.pi]]  # almost collision-free
TEMP_HARDCODE_PATH = [[2, -.5, 0], [2.4, -1, -np.pi/2], [2.45, -3.5, -np.pi/2], [1.5, -4.4, np.pi]]  # some possible collisions


#Map Handling Functions
def load_map(filename):
    import matplotlib.image as mpimg
    import cv2
    im = cv2.imread("../maps/" + filename)
    im = cv2.flip(im, 0)
    # im = mpimg.imread("../maps/" + filename)
    if len(im.shape) > 2:
        im = im[:,:,0]
    im_np = np.array(im)  #Whitespace is true, black is false
    im_np = np.logical_not(im_np)     #for ros
    return im_np

class PathFollower():
    def __init__(self):
        # time full path
        self.path_follow_start_time = rospy.Time.now()

        # use tf2 buffer to access transforms between existing frames in tf tree
        self.tf_buffer = tf2_ros.Buffer()
        self.listener = tf2_ros.TransformListener(self.tf_buffer)
        rospy.sleep(1.0)  # time to get buffer running

        # constant transforms
        self.map_odom_tf = self.tf_buffer.lookup_transform('map', 'odom', rospy.Time(0), rospy.Duration(2.0)).transform
        # print(self.map_odom_tf)

        # subscribers and publishers
        self.cmd_pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
        self.global_path_pub = rospy.Publisher('~global_path', Path, queue_size=1, latch
```

```
=True)
        self.local_path_pub = rospy.Publisher('~local_path', Path, queue_size=1)
        self.collision_marker_pub = rospy.Publisher('~collision_marker', Marker, queue_size=1)

        # map
        # map = rospy.wait_for_message('/map', OccupancyGrid)
        # self.map_np = np.array(map.data).reshape(map.info.height, map.info.width)
        # self.map_resolution = round(map.info.resolution, 5)
        # self.map_origin = -utils.se2_pose_from_pose(map.info.origin)  # negative because of weird way origin is stored
        # self.map_nonzero_idxes = np.argwhere(self.map_np)
        map_filename = "myhal.png"
        occupancy_map = load_map(map_filename)
        self.map_np = occupancy_map
        self.map_resolution = 0.05
        self.map_origin = np.array([ 0.2 , 0.2 ,-0. ])
        self.map_nonzero_idxes = np.argwhere(self.map_np)


        # collisions
        self.collision_radius_pix = COLLISION_RADIUS / self.map_resolution
        self.collision_marker = Marker()
        self.collision_marker.header.frame_id = '/map'
        self.collision_marker.ns = '/collision_radius'
        self.collision_marker.id = 0
        self.collision_marker.type = Marker.CYLINDER
        self.collision_marker.action = Marker.ADD
        self.collision_marker.scale.x = COLLISION_RADIUS * 2
        self.collision_marker.scale.y = COLLISION_RADIUS * 2
        self.collision_marker.scale.z = 1.0
        self.collision_marker.color.g = 1.0
        self.collision_marker.color.a = 0.5

        # transforms
        self.map_baselink_tf = self.tf_buffer.lookup_transform('map', 'base_footprint', rospy.Time(0), rospy.Duration(2.0))
        self.pose_in_map_np = np.zeros(3)
        self.pos_in_map_pix = np.zeros(2)
        self.update_pose()

        # path variables
        cur_dir = os.path.dirname(os.path.realpath(__file__))

        # to use the temp hardcoded paths above, switch the comment on the following two lines
        self.path_tuples = np.load(os.path.join(cur_dir, 'shortest_path.npy')).T
        # self.path_tuples = np.array(TEMP_HARDCODE_PATH)

        self.path = utils.se2_pose_list_to_path(self.path_tuples, 'map')
        self.global_path_pub.publish(self.path)

        # goal
        self.cur_goal = np.array(self.path_tuples[0])
        self.cur_path_index = 0

        # trajectory rollout tools
        # self.all_opts is a Nx2 array with all N possible combinations of the t and v vels, scaled by integration dt
        self.all_opts = np.array(np.meshgrid(TRANS_VEL_OPTS, ROT_VEL_OPTS)).T.reshape(-1, 2)

        # if there is a [0, 0] option, remove it
        all_zeros_index = (np.abs(self.all_opts) < [0.001, 0.001]).all(axis=1).nonzero()[0]
        if all_zeros_index.size > 0:
            self.all_opts = np.delete(self.all_opts, all_zeros_index, axis=0)
        self.all_opts_scaled = self.all_opts * INTEGRATION_DT

        self.num_opts = self.all_opts_scaled.shape[0]
        self.horizon_timesteps = int(np.ceil(CONTROL_HORIZON / INTEGRATION_DT))

        self.rate = rospy.Rate(CONTROL_RATE)
```

```
        self.prev_ctrl = np.array([0,0])

        rospy.on_shutdown(self.stop_robot_on_shutdown)
        self.follow_path()

    def trajectory_rollout(self, vel, rot_vel, x_0):

        # Given your chosen velocities determine the trajectory of the robot for your given timestep
        # The returned trajectory should be a series of points to check for collisions

        # x_0 is expected as x, y, theta
        # x_k_1 = x_k + B(x_k)u_k*dt
        B = lambda x_theta : np.array([[np.cos(x_theta), 0],[np.sin(x_theta), 0],[0, 1]])

        # initialize output
        trajectory = np.zeros((3, 1))

        # prepare variables needed for computation
        u = np.array([[vel],[rot_vel]])
        dt = INTEGRATION_DT
        # control is computed from the current pose
        # in the global reference frame
        currState = np.reshape(x_0, (3,1))
        nextState = np.zeros((3,1))

        for i in range(0, self.horizon_timesteps):

            # compute B(x_k)
            B_k = B(currState[2][0])
            # compute new state
            #print(np.shape(B_k))
            #print(np.shape(u))
            nextState = currState + np.dot(B_k,u)*dt

            # print("state propagation")
            # print(currState)
            # print(nextState)

            # adjust heading to be in [-pi, pi]
            if nextState[2] > np.pi:
                nextState[2] = nextState[2] - 2*np.pi
            elif nextState[2] < -np.pi:
                nextState[2] = nextState[2] + 2*np.pi

            # save current state and progress to the next timestep
            if i == 0:
                trajectory = nextState
            else:
                trajectory = np.hstack((trajectory, nextState))

            currState = nextState

        return np.transpose(trajectory)

    def points_to_robot_circle(self, points):
        #Convert a series of [x,y] points to robot map footprints for collision detection
        #Hint: The disk function is included to help you with this function

        # transform robot coordinates to cell coordinates
        # cellPos = self.point_to_cell(points)
        cellPos = points

        # create a base region for the radius specified
        radRes = int(COLLISION_RADIUS/self.map_resolution) + 1# TODO: check if self map resolution is legit
        rr, cc = disk((0, 0), radRes)
        baseCoords = np.vstack((rr,cc), dtype = np.int32)

        # create result place holder
        tempX = np.zeros((2,1))
```

```
        pts2Rob = {}

        for i in range(0, cellPos.shape[1]):

            # check for duplicates
            keyLoc = (cellPos[0][i], cellPos[1][i])

            if keyLoc not in pts2Rob:

                # extract current points
                tempX[0] = cellPos[0][i]
                tempX[1] = cellPos[1][i]
                # map the occupancy region
                tempDiff = tempX + baseCoords # shape of (2, 2*radRes + 1)
                # adjust for corner cases
                tempDiff[0, tempDiff[0] >= self.map_np.shape[1]] = self.map_np.shape[1] - 1
                tempDiff[1, tempDiff[1] >= self.map_np.shape[0]] = self.map_np.shape[0] - 1
                remCells = np.where(tempDiff < 0)[1]
                tempDiffDel = np.delete(tempDiff, remCells, axis = 1)
                # save in dictionary only unique coordinates and entries
                pts2Rob[keyLoc] = np.unique(tempDiffDel, axis = 1)

        return pts2Rob

    def checkCollision(self, points_T):
        # points are of shape (2, N) in discrete coordinates
        points = np.transpose(points_T)
        # get the robot occupancy map for each set of center points
        pts2RobPoses = self.points_to_robot_circle(points) # dictionary with occupancy regions
        # dictionary with keys being center points, and data being an array of coordinates of rob occupancy

        # check if pose is collision free
        for key in pts2RobPoses:

            # get robot occupancy map: (2, M) array
            mapOcc = pts2RobPoses[key].astype(int) # coordiantes based on set passed in by "points"
            # print("occupied cells by key:{}".format(key))
            # print(mapOcc)

            # plt.scatter(-self.map_xy[0], self.map_xy[1])
            # plt.scatter(-mapOcc[1,:], mapOcc[0,:])
            # plt.scatter(-key[1], key[0], s = 100)
            # plt.scatter(-points[1,0], points[0,0])
            # plt.show()
            # exit()

            # check if any of the environment occupancy map cells occupied by the robot contains a value of 0 and report collision as True
            crash = np.any(self.map_np[mapOcc[1, :], mapOcc[0, :]] == 100) # TODO: ensure this is how map_np works
            if crash:
                return True

        return False

    def calculate_cost(self, cand_opt, end_pt):
        # 2 preferences: low dist from goal, low change from previous ctrl
        trans_factor = 16
        rot_factor = 0
        diff_rot_factor = 0

        # difference in goal poses penalized
        curr_goal = self.cur_goal
        pose_diff = np.abs(curr_goal - end_pt)
        loc_cost = trans_factor*(pose_diff[0] + pose_diff[1])
        rot_cost = rot_factor*pose_diff[2] / loc_cost

        # change in control penalzie
```

```
        curr_trans_opt = cand_opt[0]
        last_trans_opt = self.prev_ctrl[0]
        curr_rot_opt = cand_opt[1]
        last_rot_opt = self.prev_ctrl[1]
        ctrl_chg_cost = np.abs(curr_trans_opt - last_trans_opt) + diff_rot_factor*np.abs(curr_rot_opt - last_rot_opt)

        return loc_cost + rot_cost + ctrl_chg_cost # TODO: maybe implement a saturating cost for rotational error

    def follow_path(self):
        while not rospy.is_shutdown():
            # timing for debugging...loop time should be less than 1/CONTROL_RATE
            tic = rospy.Time.now()

            self.update_pose()
            self.check_and_update_goal()

            t1 = rospy.Time.now()
            # start trajectory rollout algorithm
            local_paths = np.zeros([self.horizon_timesteps + 1, self.num_opts, 3])
            local_paths[0] = np.atleast_2d(self.pose_in_map_np).repeat(self.num_opts, axis=0)

            # propogate trajectory forward, assuming perfect control of velocity and no dynamic effects

            for i, opts in enumerate(self.all_opts):
                trans_vel = opts[0]
                rot_vel = opts[1]
                pred_traj = self.trajectory_rollout(trans_vel, rot_vel, self.pose_in_map_np)
                local_paths[1:,i,:] = pred_traj #assign complete trajectory one at a time
                # print(local_paths[:,0,:])
            # print("start")
            # print(local_paths[0,:,:])
            # print("end pts")
            # print(local_paths[-1,:,:])

            t2 = rospy.Time.now()
            traj_rollout_time = t2 - t1

            # check all trajectory points for collisions
            # first find the closest collision point in the map to each local path point
            local_paths_pixels = np.rint((self.map_origin[:2] + local_paths[:, :, :2]) / self.map_resolution).astype(int) #round to nearest int as index
            # print("local path shape")
            # print(np.shape(local_paths_pixels))
            # plt.scatter(-self.map_xy[0], self.map_xy[1])
            # plt.scatter(-local_paths_pixels[0,:,1], local_paths_pixels[0,:,0])
            # print(-local_paths_pixels[0,0,0:2])
            #
            # print(-local_paths_pixels[-1,:,0:2])
            # plt.scatter(-local_paths_pixels[-1,:,1], local_paths_pixels[-1,:,0])
            # plt.show()


            valid_opts = list(range(self.num_opts))
            invalid_opts = []
            local_paths_lowest_collision_dist = np.ones(self.num_opts) * 50

            for opt in range(local_paths_pixels.shape[1]):
                # for timestep in range(local_paths_pixels.shape[0]):

                rob_center_pixels = local_paths_pixels[:,opt,:]

                # plt.scatter(-self.map_xy[0], self.map_xy[1])
                # plt.scatter(-rob_center_pixels[:,1], rob_center_pixels[:,0])
                # plt.show()
                # print(rob_center_pixels)
                # plt.scatter(-self.map_xy[0], self.map_xy[1])
                # plt.scatter(-rob_center_pixels[:,1], rob_center_pixels[:,0])
```

```
                # plt.show()
                # print(np.shape(rob_center_pixels))

                if self.checkCollision(rob_center_pixels): # if there is a collision anywhere along timstep
                    # print(rob_center_pixels)
                    # plt.scatter(-self.map_xy[0], self.map_xy[1])
                    # plt.scatter(-rob_center_pixels[:,1], rob_center_pixels[:,0], c='red')
                    # plt.show()
                    # exit()
                    valid_opts.remove(opt) # we immediately remove the option from the lists
            t3 = rospy.Time.now()
            # collison_det_time = t3 - t2
            # remove trajectories that were deemed to have collisions

            # calculate final cost and choose best option
            # final_cost = np.zeros(self.num_opts)
            final_cost = np.zeros(len(valid_opts))
            for i in range(0, len(valid_opts)):
                # print("checking opts")
                # print(local_paths[-1,i,:])
                cur_opt = self.all_opts_scaled[i]
                final_cost[i] = self.calculate_cost(cur_opt, local_paths[-1,valid_opts[i],:])

            if final_cost.size == 0:  # hardcoded recovery if all options have collision
            # if np.count_nonzero(final_cost) == 0:
                control = [-.1, 0]
            else:
                best_opt = valid_opts[final_cost.argmin()]
                # print("chosen_control")
                # print(final_cost.argmin())
                # print(best_opt)
                # print(self.all_opts[best_opt])
                # print(local_paths[-1,best_opt,:])
                control = self.all_opts[best_opt]
                self.local_path_pub.publish(utils.se2_pose_list_to_path(local_paths[:,best_opt], 'map'))

            # t4 = rospy.Time.now()
            # cost_calc_time = t4 - t3
            # send command to robot
            self.cmd_pub.publish(utils.unicycle_vel_to_twist(control))

            # uncomment out for debugging if necessary
            # print("Selected control: {control}, Loop time: {time}, Max time: {max_time}".format(
                #  control=control, time=(rospy.Time.now() - tic).to_sec(), max_time=1/CONTROL_RATE))
            # print("traj rollout time: {}, collision det time: {}, cost calc time: {}".format(traj_rollout_time, collison_det_time, cost_calc_time))
            self.rate.sleep()

    def update_pose(self):
        # Update numpy poses with current pose using the tf_buffer
        self.map_baselink_tf = self.tf_buffer.lookup_transform('map', 'base_footprint', rospy.Time(0)).transform
        self.pose_in_map_np[:] = [self.map_baselink_tf.translation.x, self.map_baselink_tf.translation.y,
                                  utils.euler_from_ros_quat(self.map_baselink_tf.rotation)[2]]
        self.pos_in_map_pix = (self.map_origin[:2] + self.pose_in_map_np[:2]) / self.map_resolution
        self.collision_marker.header.stamp = rospy.Time.now()
        self.collision_marker.pose = utils.pose_from_se2_pose(self.pose_in_map_np)
        self.collision_marker_pub.publish(self.collision_marker)

    def check_and_update_goal(self):
        # iterate the goal if necessary
        dist_from_goal = np.linalg.norm(self.pose_in_map_np[:2] - self.cur_goal[:2])
        abs_angle_diff = np.abs(self.pose_in_map_np[2] - self.cur_goal[2])
```

```
        rot_dist_from_goal = min(np.pi * 2 - abs_angle_diff, abs_angle_diff)
        if dist_from_goal < TRANS_GOAL_TOL and rot_dist_from_goal < ROT_GOAL_TOL:
            rospy.loginfo("Goal {goal} at {pose} complete.".format(
                    goal=self.cur_path_index, pose=self.cur_goal))
            if self.cur_path_index == len(self.path_tuples) - 1:
                rospy.loginfo("Full path complete in {time}s! Path Follower node shutting down.".format(
                    time=(rospy.Time.now() - self.path_follow_start_time).to_sec()))
                rospy.signal_shutdown("Full path complete! Path Follower node shutting down.")
            else:
                self.cur_path_index += 1
                self.cur_goal = np.array(self.path_tuples[self.cur_path_index])
        else:
            rospy.logdebug("Goal {goal} at {pose}, trans error: {t_err}, rot error: {r_err}.".format(
                goal=self.cur_path_index, pose=self.cur_goal, t_err=dist_from_goal, r_err=rot_dist_from_goal
            ))

    def stop_robot_on_shutdown(self):
        self.cmd_pub.publish(Twist())
        rospy.loginfo("Published zero vel on shutdown.")


if __name__ == '__main__':
    try:
        rospy.init_node('path_follower', log_level=rospy.DEBUG)
        pf = PathFollower()
    except rospy.ROSInterruptException:
        pass
```